Chapter-6

Transaction & Recovery Management

Agenda

- Introduction to transaction
- ACID properties
- Transaction states
- Concurrency problems
- View serializability and conflicts
- Recoverability and cascadlessness

Introduction to transaction

- A transaction is a **set of instructions** which performs logical unit of transactions on database, which might **modify the contents of database.**
- Every transaction should be **atomic in nature**, i.e. necessarily all the instructions within that transactions must be executed **fully or none** of them must be executed.
- The database must **remain in consistent state**, all the time.

Transaction/ACID Properties

- **A** - **Atomicity** : The **execution of entire transaction** should take place at **once or not at all.**
- **C** - **Consistency** : The database **must remain consistent before and after the execution of transaction.**
- **I** - **Isolation** : **Transactions executing concurrently** should **remain independent** and **not get affected** by each other.
- **D** - **Durability** : The **successful transactions are reflected even** if there is **system failure**.

Atomicity

- Either the entire transaction takes place at **once or doesn't happen at all**, i.e. the **transaction never occurs partially.**
- Each transaction is considered as **one unit and either runs to completion** or is **not executed at all**. It involves the following two operations:
- **- Abort**: If a transaction aborts, changes made to database are not visible.
  - Commit: If a transaction commits, changes made are visible.
- Atomicity is also known as the '**All or nothing rule'**.

Consistency

- The **database must remain in a consistent state** after any transaction.
- **No transaction** should have any **adverse effect** on the data residing in the database.
- If the database was in **a consistent state before the execution of a transaction, it must remain consistent after the execution of the transaction as well.**

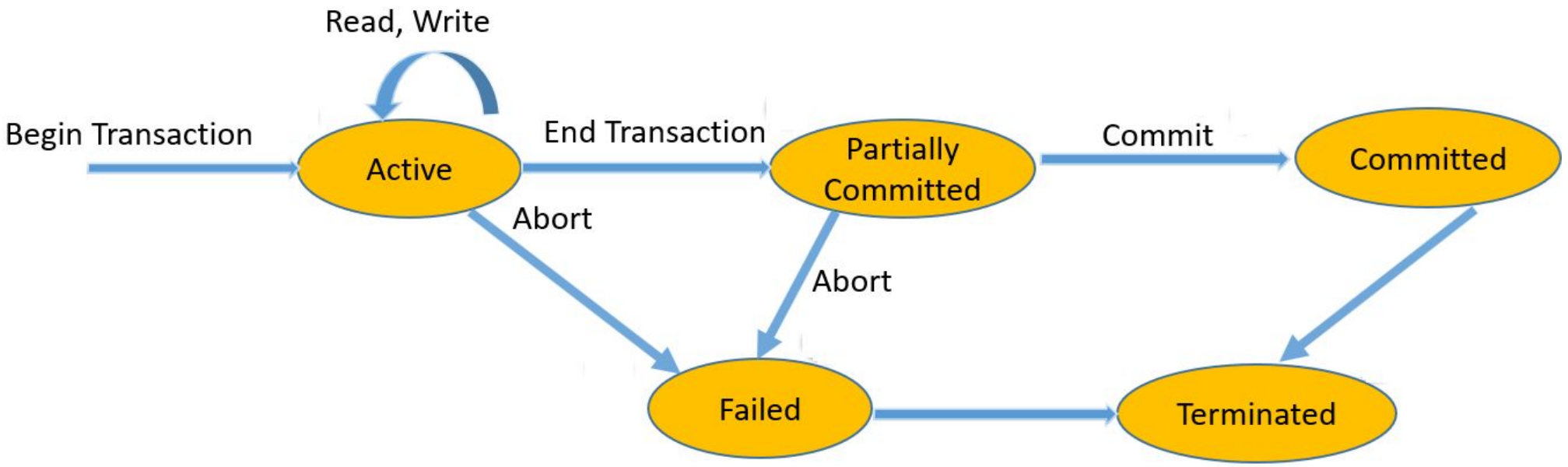
Isolation

- In a database system where **more than one transaction are being executed simultaneously** and in parallel, the property of isolation states that all the transactions will be **carried out** and executed as if it is the **only transaction** in the system is **being executed.**
- No transaction should **affect the existence** of any other transaction.

Durability

- The database should **be durable enough to hold all its latest updates** even if **the system fails or restarts.**
- If a **transaction updates a chunk of data in a database** and **commits, then the database will hold the modified data.**
- If **a transaction commits** but the **system fails before the data** could be **written on to the disk**, then that **data will be updated once the system springs back into action.**

Transaction States Diagram



Serializability of transactions

- When multiple transactions are **running concurrently** then there is a **possibility** that the **database may be left in an inconsistent state**.
- Serializability is a concept that helps us to check which **schedules** are serializable.
- A serializable schedule is one that always leaves the database in a consistent state.
- **Schedule** – A chronological execution sequence of a transaction is called a schedule. **A schedule can have many transactions** in it, each comprising a **number of instructions/tasks.**

What is a serialized schedule?

- A serializable schedule always leaves the database in consistent state.
- A **serial schedule** is always a serializable schedule because in serial schedule, a **transaction only starts** when the **other transaction finished execution.**

What is non serial schedule?

- A non-serial schedule of n number of transactions is **executed based on concurrency** but in a serializable manner, so that might be considered as serial execution of n transactions.

- A serial schedule **doesn't allow concurrency,** only one transaction **executes at a time** and the **other starts when the already running transactions n finished.**

Serializability of transactions(cont..)

- The **execution sequence** of an instruction in a transaction **cannot be changed**, but two transactions can have **their instructions executed in a random fashion.**
- **Non-serial schedule: It** defines the operations from a **group of concurrent transactions that are interleaved**.
- In non-serial schedule, **if the schedule is not proper**, then the **problems can arise like multiple update, uncommitted dependency and incorrect analysis.**
- This execution does **not harm if two transactions are mutually independent** and working on **different segments of data**; but in case these two transactions are **working on the same data**, then the **results may vary.** This ever-varying result may bring the database to an inconsistent state.

Two types of serializability:

1. **Conflict Serializability**
2. **View Serializability**

Conflict Serializability

- **Conflict Serializability** is one of the type of Serializability, which can be used to check whether a non-serial schedule is conflict serializable or not.
- A schedule is called conflict serializable if we can convert it into a serial schedule after swapping its non-conflicting operations.

Conflicting operations

- Two operations are said to be in conflict, if they satisfy all the following three conditions:
  1. Both the operations should **belong to different transactions.**
  2. Both the **operations are working on same data item.**
  3. At least one of the operation **is a write operation.**

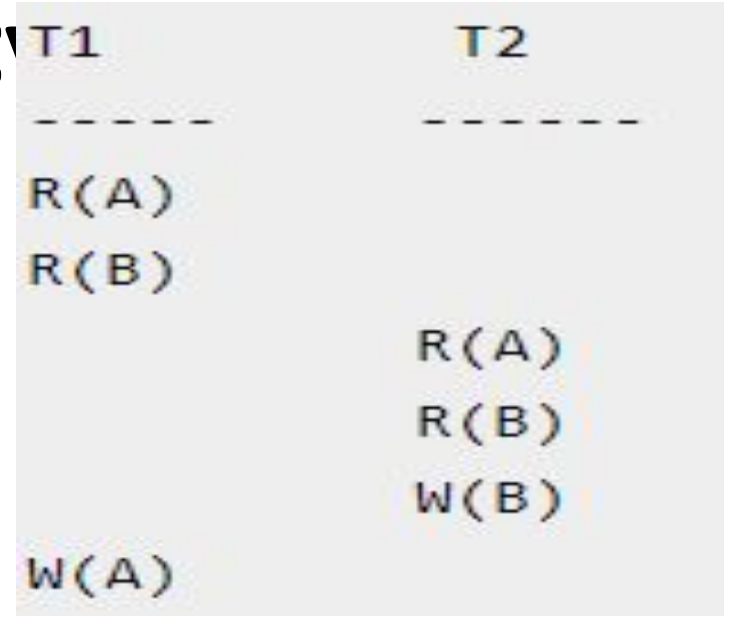
Example 1: Operation W(X) of transaction T1 and operation R(X) of transaction T2 are conflicting operations, because they satisfy all the three conditions mentioned above. They belong to different transactions, they are working on same data item X, one of the operation in write operation.

Examples of conflicting operations

- **Example 2:** Similarly Operations W(X) of T1 and W(X) of T2 are conflicting operations.
- **Example 3:** Operations W(X) of T1 and W(Y) of T2 are **non-conflicting** operations because both the write operations are not working on same data item so these operations don't satisfy the second condition.
- **Example 4:** Similarly R(X) of T1 and R(X) of T2 are **non-conflicting** operations because none of them is write operation.
- **Example 5:** Similarly W(X) of T1 and R(X) of T1 are non-conflicting operations because both the operations belong to same transaction T1.

Example of Conflict Serializability

- Lets consider this schedule:



- To convert this schedule into a serial schedule we must have to swap the R(A) operation of transaction T2 with the W(A) operation of transaction T1. However we cannot swap these two operations because they are conflicting operations, thus we can say that this given schedule is **not Serializable**.

View Serializability

- View Serializability is a process to find out that a given **schedule** is **view serializable or not.**
- To check whether a **given schedule is view serializable**, we need to check whether the given schedule is **View Equivalent** to its serial schedule.

View Serializability(Example)

- In **Serial schedule** a transaction only starts when the **current running transaction is finished.** So the serial schedule of the above given schedule would look like this:

T1	T2
R(X)	
W(X)	
	R(X)
	W(X)
R(Y)	
W(Y)	
	R(Y)
	W(Y)

T1	T2
R(X)	
W(X)	
R(Y)	
W(Y)	
	R(X)
	W(X)
	R(Y)
	W(Y)

3 Rules to check view equivalency

- 1. Initial Read
  - An initial read of both schedules must be the same. Suppose two schedule S1 and S2. In schedule S1, if a transaction T1 is reading the data item A, then in S2, transaction T1 should also read A.

T1	T2
Read(A)	
	Write(A)

Schedule S1

T1	T2
	Write(A)
Read(A)	

Schedule S2

  - two schedules are view equivalent because the Initial read operation in S1 is done by T1 and in S2 it is also done by T1.

2. Updated Read

- In schedule S1, if Ti is reading A which is updated by Tj then in S2 also, Ti should read A which is updated by Tj.

T1	T2	T3
Write(A)		
	Write(A)	
		Read(A)

Schedule S1

T1	T2	T3
	Write(A)	
Write(A)		
		Read(A)

Schedule S2

- two schedules are not view equal because, in S1, T3 is reading A updated by T2 and in S2, T3 is reading A updated by T1.

3 Rules to check view equivalency(cont.)

3. Final Write
   - A final write must be the same between both the schedules. In schedule S1, if a transaction T1 updates A at last then in S2, final writes operations should also be done by T1.

T1	T2	T3
Write(A)	Read(A)	Write(A)

Schedule S1

T1	T2	T3
Write(A)	Read(A)	Write(A)

Schedule S2

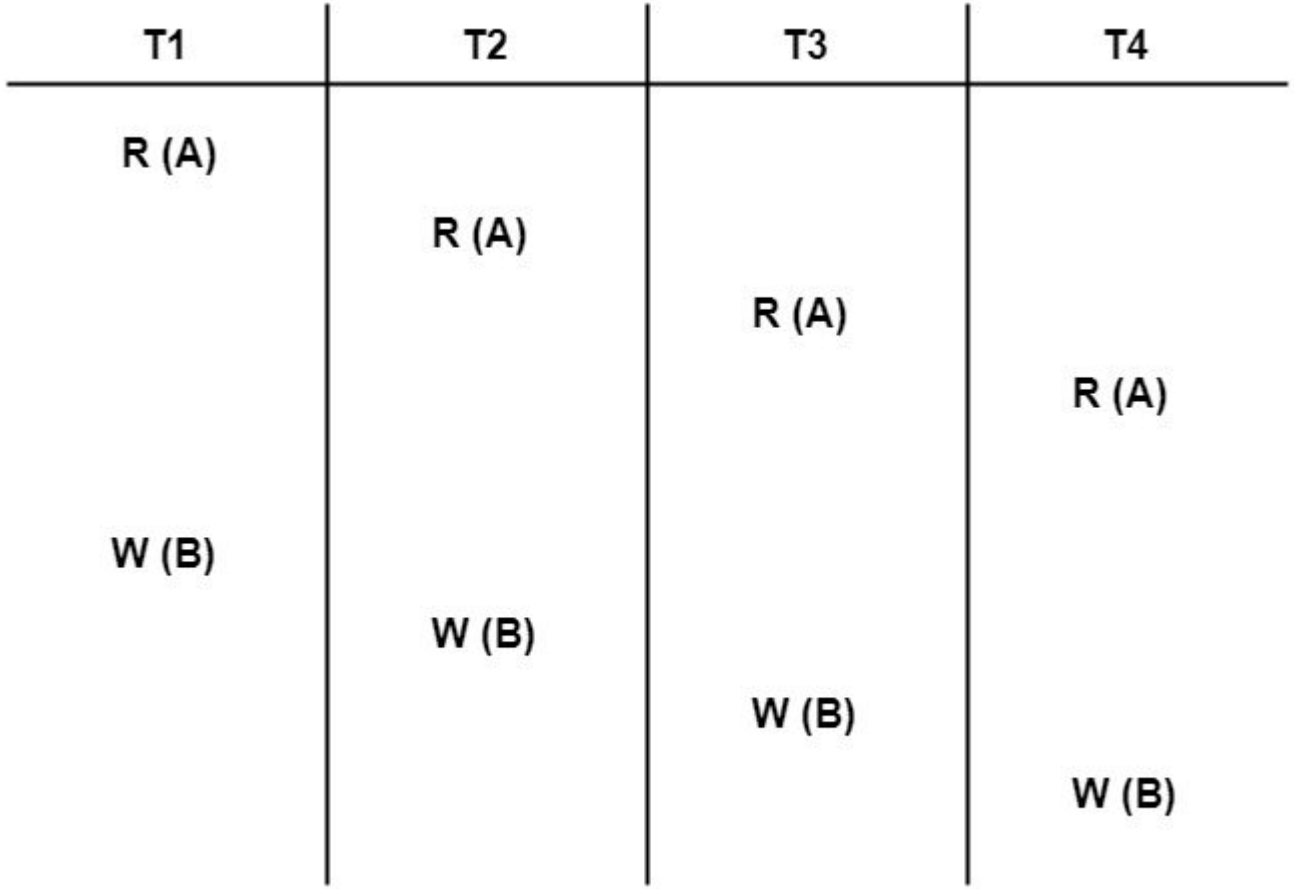
   - two schedules is view equal because Final write operation in S1 is done by T3 and in S2, the final write operation is also done by T3.

Need for View Serializability

- Serial schedule never leaves the **database in inconsistent state** because there are **no concurrent transactions execution.**
- While **a non-serial schedule can leave the database in inconsistent state** because there are **multiple transactions** running **concurrently.**
- By verifying that a **given non-serial schedule is view serializable**, we can make sure that it is a **consistent schedule.**

Example

- Check whether the given schedule S is view serializable or not.



Solution

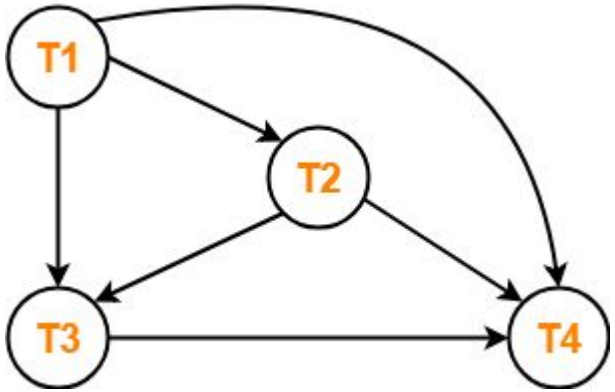
- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

Checking Whether S is Conflict Serializable Or Not-

- **Step-01:**

- List all the conflicting operations and determine the dependency between the transactions-
  - $W_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
  - $W_1(B)$, $W_3(B)$ $(T_1 \rightarrow T_3)$
  - $W_1(B)$, $W_4(B)$ $(T_1 \rightarrow T_4)$
  - $W_2(B)$, $W_3(B)$ $(T_2 \rightarrow T_3)$
  - $W_2(B)$, $W_4(B)$ $(T_2 \rightarrow T_4)$
  - $W_3(B)$, $W_4(B)$ $(T_3 \rightarrow T_4)$

- **Step-02:**
  - Draw the precedence graph



- Clearly, there exists no cycle in the precedence graph.
- Therefore, the given schedule S is conflict serializable.
- Thus, we conclude that the given schedule is also **view serializable**.

Problem – 2

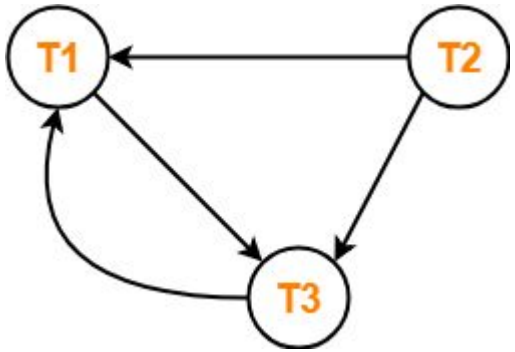
- Check whether the given schedule S is view serializable or not

T1	T2	T3
R (A)		
	R (A)	
		W (A)
W (A)		

Checking Whether S is Conflict Serializable Or Not

- List all the conflicting operations and determine the dependency between the transactions-
  - $R_1(A)$, $W_3(A)$ $(T_1 \rightarrow T_3)$
  - $R_2(A)$, $W_3(A)$ $(T_2 \rightarrow T_3)$
  - $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
  - $W_3(A)$, $W_1(A)$ $(T_3 \rightarrow T_1)$

- **Step-02:**
  - Draw the precedence graph



  - Clearly, there exists a cycle in the precedence graph.
  - Therefore, the given schedule S is not conflict serializable.

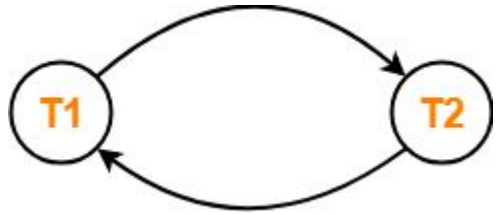
Problem-03:

T1	T2
R (A)	
A = A + 10	
	R (A)
	A = A + 10
W (A)	
	W (A)
R (B)	
B = B + 20	
	R (B)
	B = B x 1.1
W (B)	
	W (B)

- **Step-01:**

- List all the conflicting operations and determine the dependency between the transactions-
  - $R_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
  - $R_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
  - $W_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
  - $R_1(B)$, $W_2(B)$ $(T_1 \rightarrow T_2)$
  - $R_2(B)$, $W_1(B)$ $(T_2 \rightarrow T_1)$

- **Step-02:**
  - Draw the precedence graph-

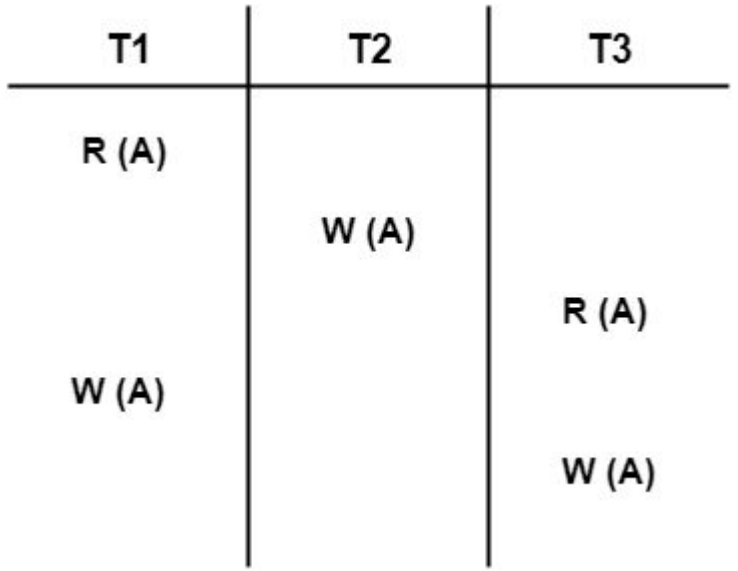


  - Clearly, there **exists a cycle** in the precedence graph.
  - Therefore, the given schedule S is **not conflict serializable**.

Problem-04:

- Check whether the given schedule S is view serializable or not. If yes, then give the serial schedule.
  - **S : $R_1(A)$, $W_2(A)$, $R_3(A)$, $W_1(A)$, $W_3(A)$**

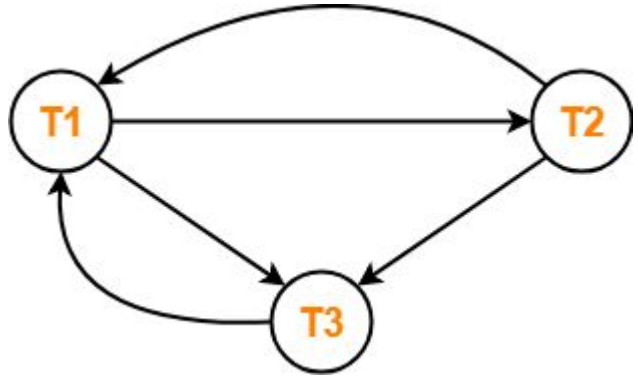
For simplicity and better understanding, we can represent the given schedule pictorially as-



- We know, if a schedule is conflict serializable, then it is surely view serializable.
- So, let us check whether the given schedule is conflict serializable or not.

- **Step-01:**
- List all the conflicting operations and determine the dependency between the transactions-
  - $R_1(A)$, $W_2(A)$ $(T_1 \rightarrow T_2)$
  - $R_1(A)$, $W_3(A)$ $(T_1 \rightarrow T_3)$
  - $W_2(A)$, $R_3(A)$ $(T_2 \rightarrow T_3)$
  - $W_2(A)$, $W_1(A)$ $(T_2 \rightarrow T_1)$
  - $W_2(A)$, $W_3(A)$ $(T_2 \rightarrow T_3)$
  - $R_3(A)$, $W_1(A)$ $(T_3 \rightarrow T_1)$
  - $W_1(A)$, $W_3(A)$ $(T_1 \rightarrow T_3)$

- **Step-02:**
  - Draw the precedence graph

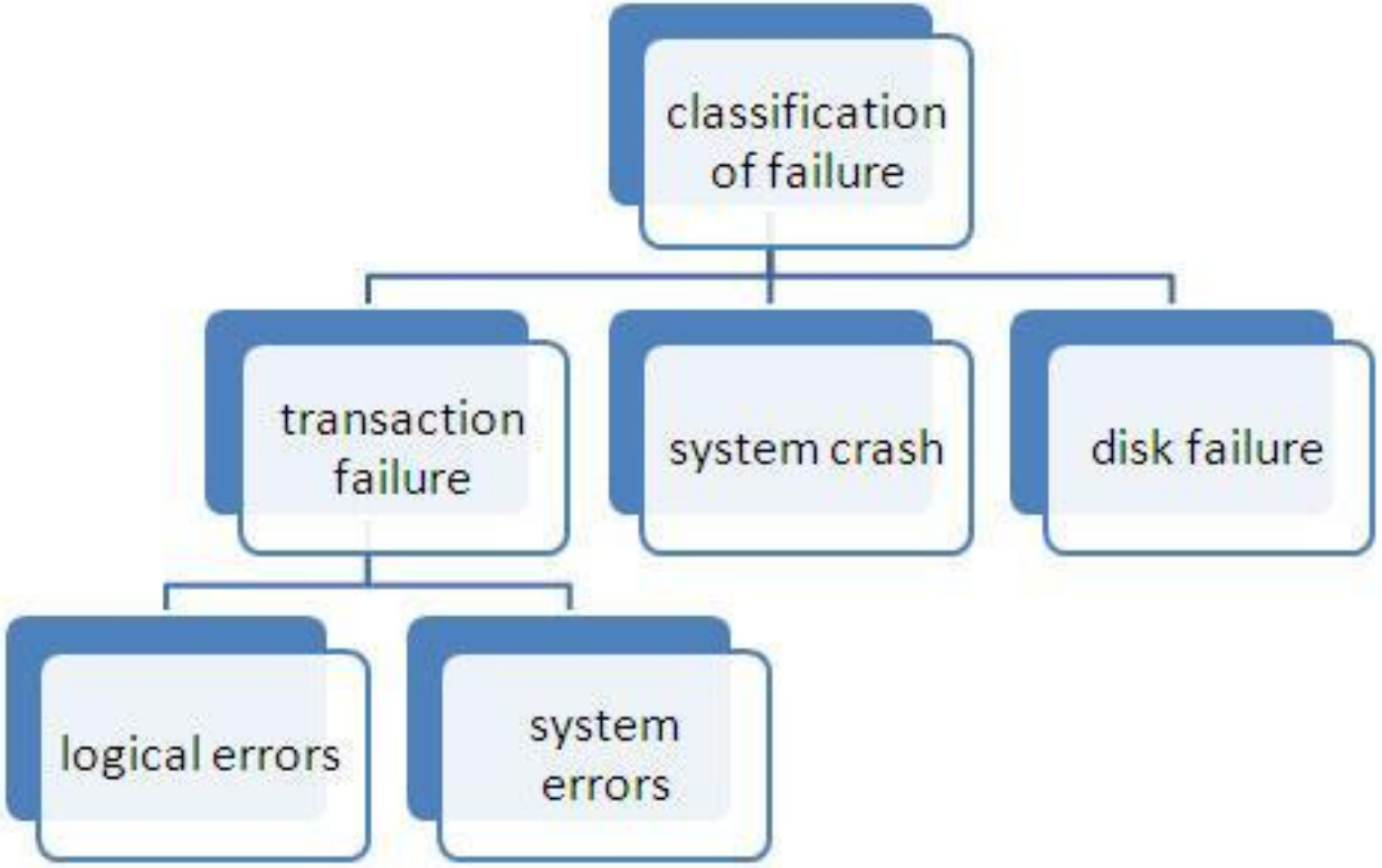


  - Clearly, there exists a cycle in the precedence graph.
  - Therefore, the given schedule S is not conflict serializable.

Database Recovery

- Database failure can occur at anytime. So, it is very necessary **that data must be made available all the time.**
- Database recovery means recovering the data when it **gets deleted, hacked or damaged accidently.**
- Atomicity and consistency of transaction must be preserved in any case, if the transaction is reflected on the database permanently or not at all.

Type of failure in DBMS



Types of failures

- **1. Transaction failure:** A transaction **needs to be aborted** once it fails to execute particular instruction or reaches to any further extent from where it cannot bring the database in consistent state.
  - Reasons for such transaction failure are:
  - **Logical errors:** Due to some internal condition or bad input, transaction cannot be continued.**(eg: division by 0)**
  - **System errors:** System has entered an undesirable state due to shortage of resource, and so the healthy transaction needs to be aborted**.(eg. deadlock)**

Types of failures

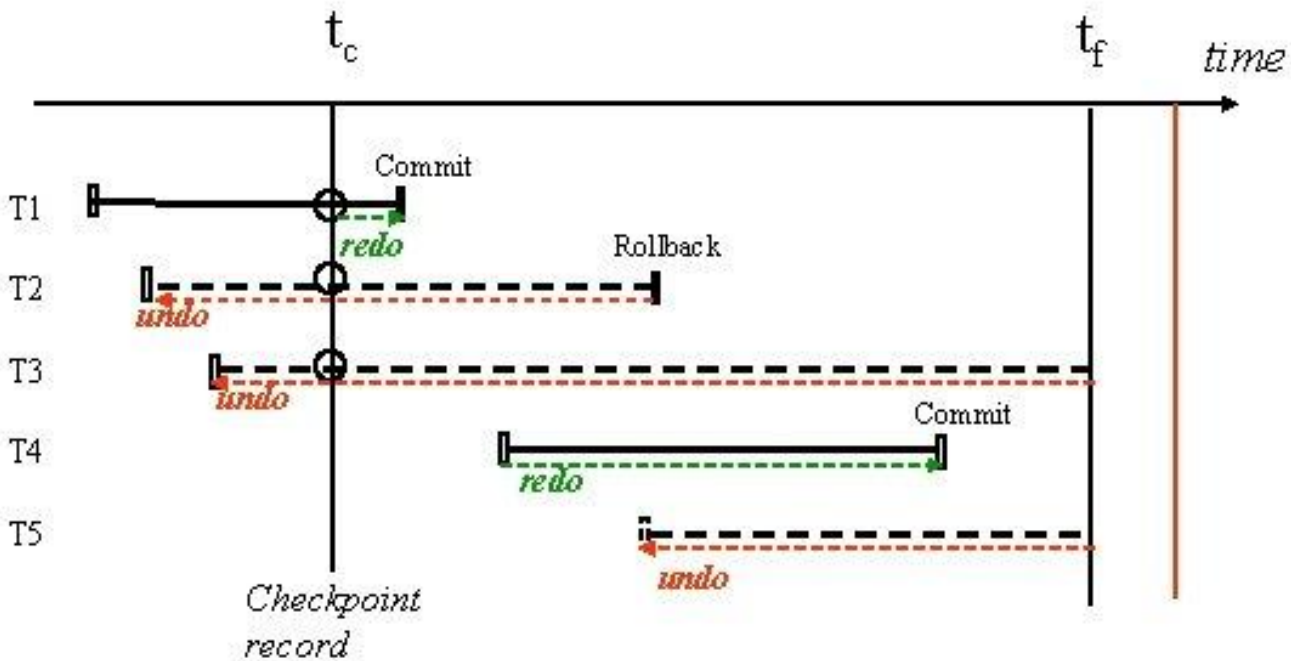
- **2. System Crash:** There is hardware malfunction, or a bug in the database software or OS, that can cause loss of volatile storage.

- **3. Disk failure:** A disk block loses its content due to head crash or failure during a data transfer operation. Taking regular backup can help to recover from such failures.

Example for failure

- Consider transaction T_i that transfers \$50 from account A to account B
  - Two updates: subtract 50 from A and add 50 to B
- Transaction T_i requires updates to A and B to be output to the database.
  - A failure may occur after **one of these modifications have been made** but before both of them are made.
  - Modifying the database **without ensuring that the transaction will commit** may leave the database in an inconsistent state
  - **Not modifying** the database **may result in lost updates** if failure occurs just after transaction commits

Rollback recovery using transaction log



Rollback Recovery

Undo list: ~~T1~~, T2, T3, ~~T4~~, T5

Redo list: T1, T4

5. **Rollback transactions** of the Undo list
   - writing the before images into the database

   Redo transactions of the Redo list
   - writing the after images into the database

6. Open the DBMS service to applications

Recovery Algorithm(cont.)

- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to **ensure enough information exists to recover from failures**
  2. Actions taken **after a failure to recover the database contents** to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

- To ensure **atomicity despite failures**, we first output information describing the modifications to stable storage without modifying the database itself.
- Log-based recovery mechanism
  - **Immediate database modification**
  - **Deferred database modification**

Log-based recovery

- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a **record of update activities** on the database.
- When transaction T_i starts, it registers itself by writing a
  $<T_i$ **start**>log record
- *Before* T_i executes **write**(X), a log record
  $<T_i, X, V_1, V_2>$
  is written, where V_1 is the value of X before the write (the **old value**), and V_2 is the value to be written to X (the **new value**).
- When T_i finishes it last statement, the log record $<T_i$ **commi**t> is written.

Immediate Database modification

- The **immediate-modification** scheme allows **updates of an uncommitted transaction to be made to the buffer**, or the disk itself, before the transaction commits
- Update log record must be written ***before*** **database item** is written on actual database.
- A write operations has both new and old value.
- **Undo and redo both are possible**.

Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

(b) redo (T_0) and undo (T_1): A and B are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600.

Deferred database modification

- The **deferred-modification** scheme performs **updates to buffer/disk only at the time of transaction commit**
- A write operation has only the new value
- Only **redo is possible** in deferred database modification.

Checkpoints

- **Redoing/undoing all transactions** recorded in the log **can be very slow**
  1. **Processing the entire log is time-consuming** if the system has run for a long time
  2. We might **unnecessarily redo transactions** which have already output their updates to the database.
- Streamline recovery procedure by **periodically performing checkpointing**
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint** *L*> onto stable storage where *L* is a list of all transactions active at the time of checkpoint.
     - All updates are stopped while doing checkpointing.

Checkpoints(cont..)

- **During recovery** we need to consider only the **most recent transaction T_i** that **started before the checkpoint**, and transactions that started after T_i.
  1. **Scan backwards** from end of log to **find the most recent <checkpoint *L*>** record
     - Only transactions that **are in *L*** or **started after the checkpoint need to be redone** or undone
     - Transactions that **committed or aborted before** the checkpoint already have all their updates output to stable storage.
- Some **earlier part of the log** may be needed for **undo operations**
  1. Continue scanning **backwards till a record** $<T_i$ **start>** is found for every transaction T_i in L.
     - Parts of log prior to earliest $<T_i$ **start>** record above are not needed for recovery, and can be erased whenever desired.

Recovery algorithm

- **Logging** (during normal operation):
  - $<T_i$ **start**$>$ at transaction start
  - $<T_i, X_j, V_1, V_2>$ for each update, and
  - $<T_i$ **commit**$>$ at transaction end
- **Transaction rollback (during normal operation)**
  - Let T_i be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of T_i of the form $<T_i, X_j, V_1, V_2>$
    - perform the undo by writing V_1 to X_j,
    - write a log record $<T_i, X_j, V_1>$
      - such log records are called **compensation log records**
  - Once the record $<T_i$ **start**$>$ is found stop the scan and write the log record $<T_i$ **abort**$>$

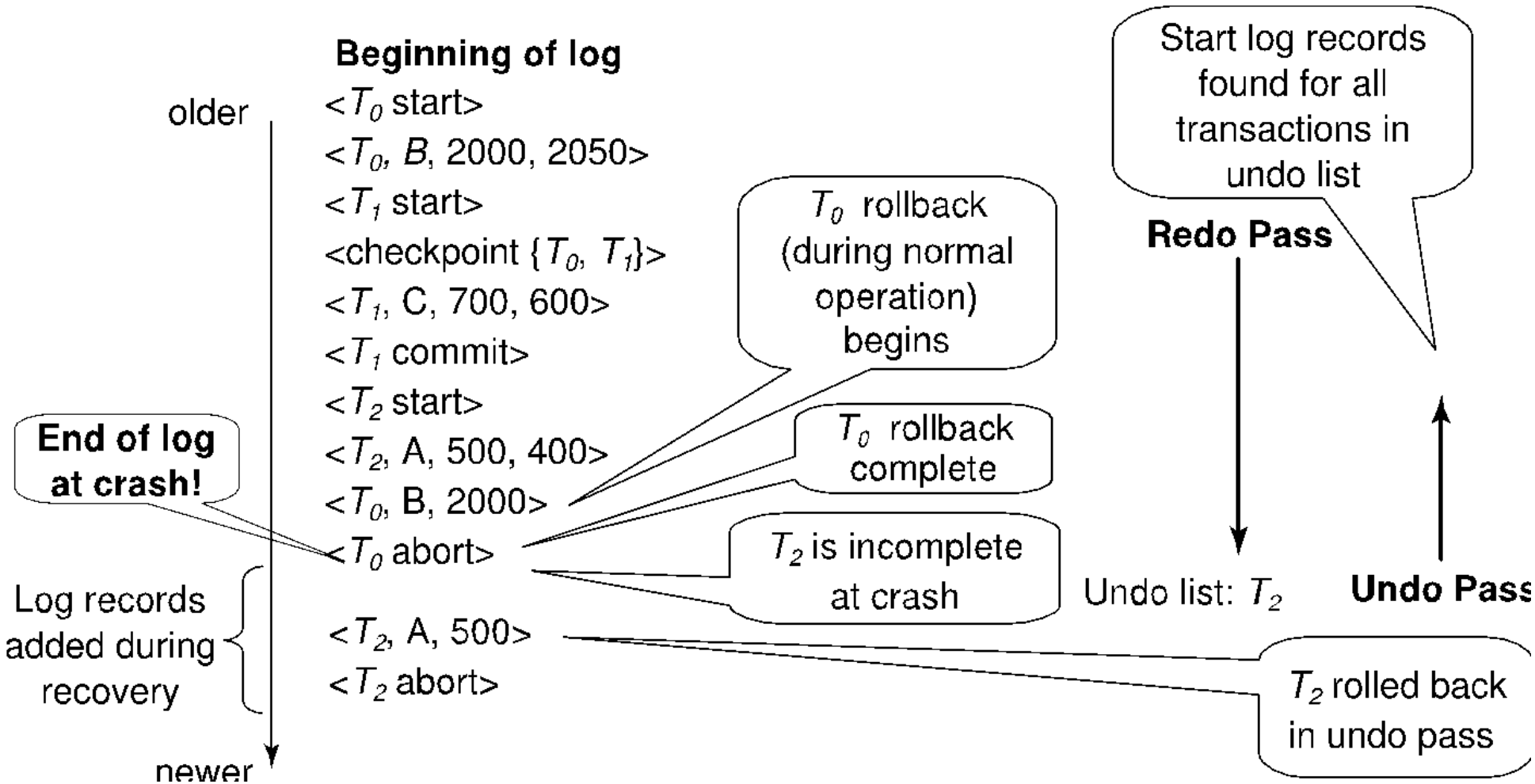
Recovery algorithm

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last<**checkpoint** L> record, and set undo-list to L.
  2. Scan forward from above <**checkpoint** L> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ is found, **redo** it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recovery algorithm(cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record $<T_i, X_j, V_1>$
     2. Whenever a log record $<T_i$ **start**$>$ is found where T_i is in undo-list,
        1. Write a log record $<T_i$ **abort**$>$
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. $<T_i$ **start**$>$ has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of recovery

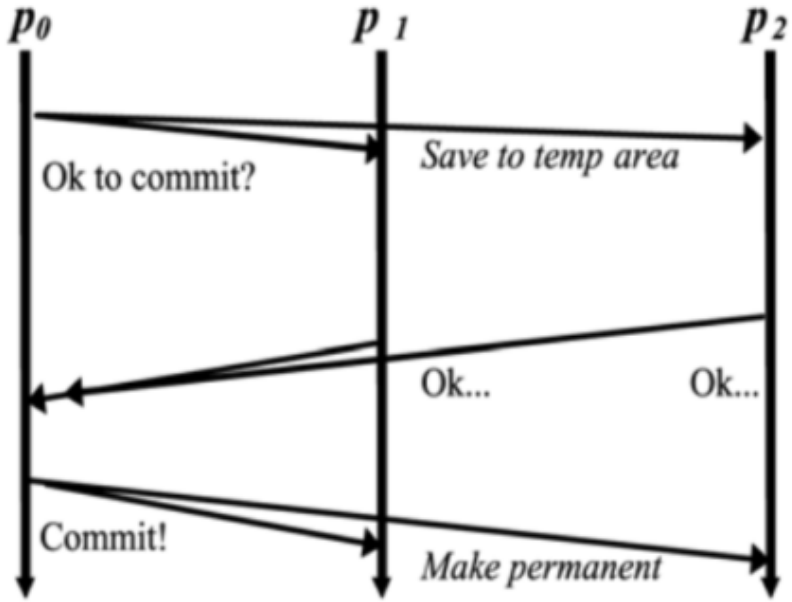


Two phase commit protocol

- It has two phases:
- **Prepare and decision**
- **In prepare phase,** a coordinator (the node initiating the transaction) makes a request to all the participating nodes, asking them whether they are able to commit the transaction.
- **So, participants will respond yes or no to the coordinator.**
- If all the participating nodes said yes, then commit request is sent to all the nodes
- If any of the participating nodes said no, then abort request is sent to all the nodes

Two phase commit protocol

- In second phase, coordinator decides based on the votes whether to send the commit or abort request to the participating nodes. **This phase is called the commit phase.**



Coordinator :
multicast: *ok to commit?*
collect replies
all *ok* => *send commit*
else => *send abort*
Participant:
ok to commit =>
save to temp area, reply *ok*
commit =>
make change permanent
abort =>
delete temp area

Two phase commit protocol

- Every transaction is given **unique identifier.**
- A prepare request is sent to all the participants by coordinator. If any request **fails or timesout**, the **coordinator sends an abort request**.
- If the prepare request is successfully received by the participating nodes then participating nodes will commit the transaction in all situations. The participant node return yes to the coordinator if everything is fine.
- Once coordinator has received all the responses to prepare requests, it makes a decision on whether to commit or abort a transaction based on the received votes.

Two phase commit protocol

- Coordinator sends the commit or abort request to all the participating nodes.
- Coordinator will enforce the decision by retrying the request if some of the requests fail. If the participating node died before committing the transaction then the transaction will be committed after it has recovered.

Concurrency control

- It is the procedure for **managing simultaneous** operations without conflicting with each other.
- Concurrency is a challenge when **any database is dealing with multiple read and write operations** on it.
- Concurrency control is very important element, when **two or multiple transactions require to access the same data simultaneously.**

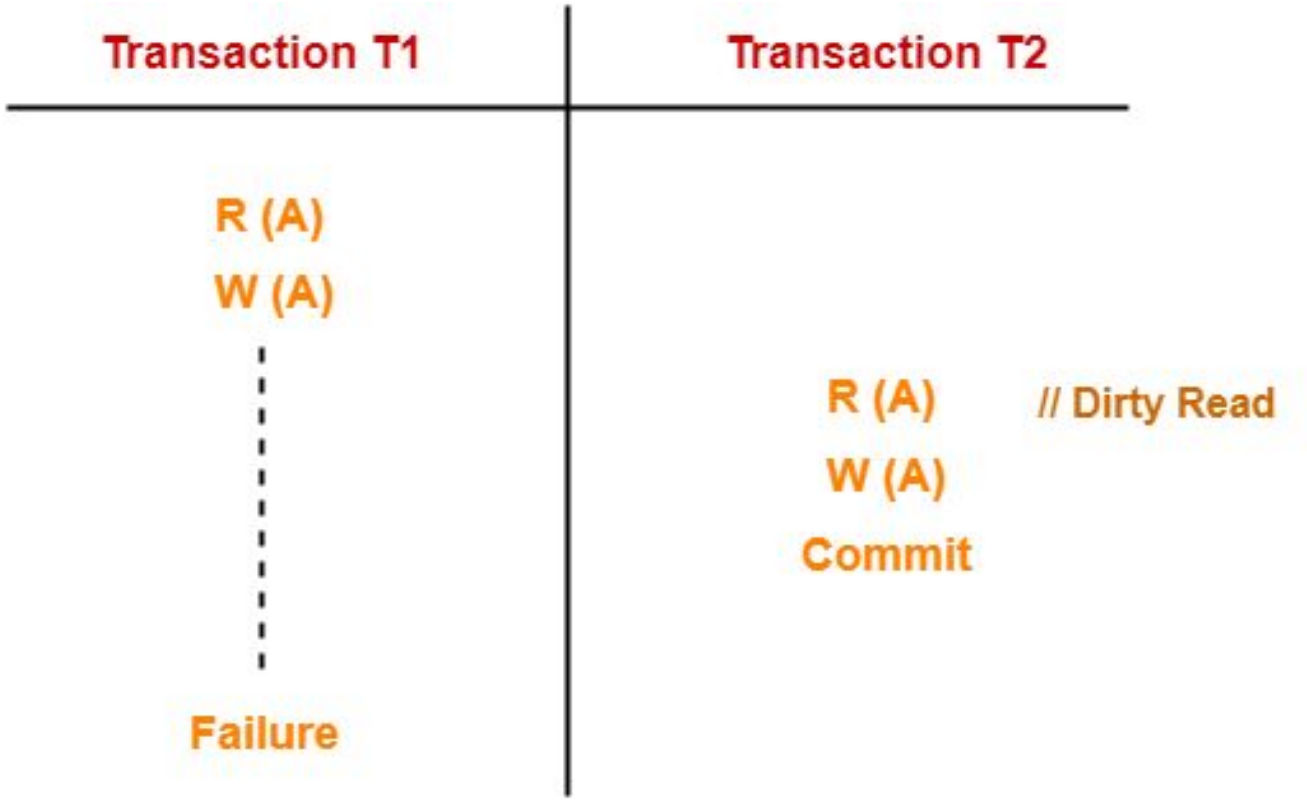
Concurrency Problem

- When multiple transactions execute concurrently in an uncontrolled or unrestricted manner, then it might lead to several problems, this problems are called concurrency problems.

Dirty read problem

- Reading the data written by an uncommitted transaction is called as dirty read.
- Reasons:
  - There is always a chance that the uncommitted transaction might roll back later.
  - Thus, uncommitted transaction might make other transactions read a value that does not even exist.
  - This leads to inconsistency of the database.

1. T1 reads the value of A.
2. T1 updates the value of A in the buffer.
3. T2 reads the value of A from the buffer.
4. T2 writes the updated the value of A.
5. T2 commits.
6. T1 fails in later stages and rolls back



In this example,

- T2 reads the dirty value of A written by the uncommitted transaction T1.
- T1 fails in later stages and roll backs.
- Thus, the value that T2 read now stands to be incorrect.
- Therefore, database becomes inconsistent.

Unrepeatable Read Problem

- This problem occurs when a transaction **gets to read unrepeated** i.e. different values of the same variable in its different read operations even when it has not updated its value.
- Two reads on same data in same transaction reads out different values.

Transaction T1	Transaction T2	
R (X)		
	R (X)	
W (X)		
	R (X)	// Unrepeated Read

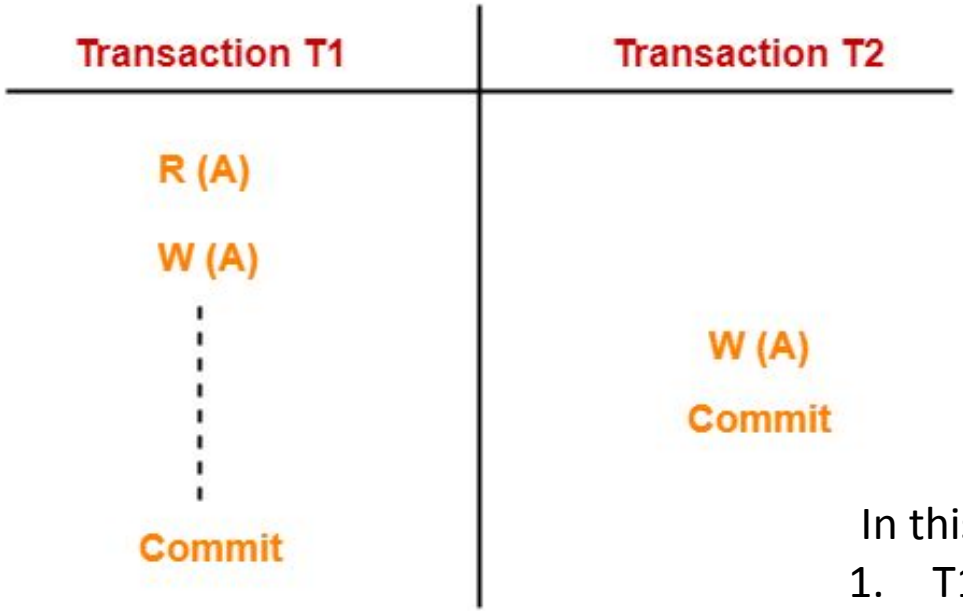
1. T1 reads the value of X (= 10 say).
2. T2 reads the value of X (= 10).
3. T1 updates the value of X (from 10 to 15 say) in the buffer.
4. T2 again reads the value of X (but = 15).

In this Example

1. T2 gets to read a different value of X in its second reading.
2. T2 wonders how the value of X got changed because according to it, it is running in isolation.

Lost update problem

- This problem occurs when multiple transactions execute concurrently and updates from one or more transactions get lost.
- This problem occurs whenever there is **a write-write conflict.**
- In write-write conflict, there are two writes one by each transaction on the same data item **without any read in the middle**.



1. T1 reads the value of A (= 10 say).
2. T2 updates the value to A (= 15 say) in the buffer.
3. T2 does blind write A = 25 (write without read) in the buffer.
4. T2 commits.
5. When T1 commits, it writes A = 25 in the database.

In this example,

1. T1 writes the over written value of X in the database.
2. Thus, update from T1 gets lost.

NOTE-

- This problem occurs whenever there is a write-write conflict.
- In write-write conflict, there are two writes one by each transaction on the same data item without any read in the middle.

Phantom Read Problem

- This problem occurs when a transaction reads some variable from the buffer and when it reads the same variable later, it finds that the variable does not exist.

Transaction T1	Transaction T2
R (X)	
	R (X)
Delete (X)	
	Read (X)

1. T1 reads X.
2. T2 reads X.
3. T1 deletes X.
4. T2 tries reading X but does not find it.

In this example,

- T2 finds that there does not exist any variable X when it tries reading X again.
- T2 wonders who deleted the variable X because according to it, it is running in isolation.

Locking mechanisms

- Locks help synchronize access to the database items by concurrent transactions.
- All lock requests are made to the concurrency-control manager. Transactions proceed only once the lock request is granted.
- 1. Exclusive (X) mode. Using lock-x instruction, the data item can be read as well as written. After finishing write operation, the data item can be unlocked.
- 2. Shared (S) mode. It is read only lock. In this, transaction will not have permission to update data. (lock-S instruction)
- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Starvation

- Starvation is the situation when **a transaction needs to wait for an indefinite period to acquire a lock.**
- Following are the reasons for Starvation:
  - When **waiting scheme for locked items is not properly managed**
  - In the case of **resource leak**
  - The **same transaction** is selected as a **victim repeatedly**

Deadlock

- Deadlock refers to a specific situation where two or more processes are waiting for each other to release a resource or more than two processes are waiting for the resource in a circular chain.